ERDC MSRC PET Technical Report No. 01-09

Developing Multi-Threaded Fortran Applications Using the PARSA Software Development Environment

by

Jeff Marquis
Geoffrey Wossum

30 April 2001

Work funded by the Department of Defense
High Performance Computing Modernization Program
U.S. Army Engineer Research and Development Center
Major Shared Resource Center through

Programming Environment and Training

Supported by Contract Number: DAHC94-96-C0002
Computer Sciences Corporation

Developing Multi-Threaded Fortran Applications Using the PARSA™ Software Development Environment

Jeff Marquis and Geoffrey Wossum

Prism Parallel Technologies, Inc.
835 East Lamar Boulevard #161
Arlington, TX 76011
www.PrismPTI.com
marquis@PrismPTI.com

The University of Texas at Arlington
Computer Science and Engineering Department
P.O. Box 19015
Arlington, TX 76019
www.cse2.uta.edu
gpw0341@omega.uta.edu

1 Introduction

This paper presents how multi-threaded Fortran applications are developed using the PARSA™ Software Development Environment. The PARSA programming methodology and its associated tools give DoD HPC users an easy-to-use, fast and efficient method for developing Fortran projects that exploit the resources of shared memory systems.

The paper begins with a brief introduction to threading, and how threads efficiently exploit the resources of shared memory systems. The paper then states that developing multi-threaded projects using directives-based programming methods puts additional development requirements on programmers, driving up the cost, time and expertise needed to develop software projects.

This is followed by an introduction to the PARSA programming methodology, which does not put additional development requirements on software projects but produces scalable and efficient multi-threaded software. The PARSA programming methodology is then shown to allow programmers to exploit many different forms of parallelism using different types of graphical objects supported by PARSA. This is followed with an in-depth presentation of the PARSA source code generator and the ThreadMan™ Thread Manager, two integral components of the PARSA programming methodology. A Fortran-Pthreads application programming interface (API) is introduced that allows PARSA programmers to embed Fortran-like threading directives into their Fortran projects if needed. An example Fortran project is then presented that demonstrates how easily multi-threaded Fortran projects can be developed in PARSA. Finally, a summary is presented and conclusions are drawn.

2 Multi-Threading

Threading is an effective way to exploit the resources of shared memory computer systems. Applications can spawn threads to execute concurrently on shared memory systems. If the system executing the application has multiple processors, the threads will be distributed by the operating system to execute on different processors resulting in improved run time performance. Further, applications can spawn more threads than the system has processors without significant performance degradations because threads are lightweight processes that efficiently time share resources. Further still, data can be passed to threads in an efficient manner utilizing the shared memory resources of the system. Hence, threading is an effective and efficient method for exploiting the resources of shared memory systems.

A more programmer-friendly way of developing multi-threaded Fortran applications is supported by the PARSA Software Development Environment. PARSA programmers are abstracted from the low-level details of threading while producing the scalable run time benefits of multi-threaded software.

3 The PARSA Programming Methodology

The PARSA programming methodology [4, 5, 6] is a unique graphical programming method that allows multi-threaded software to be developed quickly and easily. The PARSA programming methodology is based on object-based programming principals to facilitate the modular development of multi-threaded software projects so they can execute safely and reliably on multi-threaded systems. Developing multi-threaded software has unique requirements, and the PARSA programming methodology specifically meets these unique requirements.

Projects developed in PARSA consist of graphical objects (or GOs) and arcs. When a new project is being developed in PARSA graphical objects are added to the project. Each graphical object represents a project task to be performed. When a graphical object is added to a project, the interface the graphical object will have with other project graphical objects must be defined. This is done via the GO Properties Panel; each type of graphical object has unique properties that are discussed briefly below. Regardless of which type of graphical object is being added to a project, the interface properties define the interface "contract" that a graphical object has with other graphical objects in a project. Specifically, the interface defines the data an object is dependent on for execution (referred to as INPUT variables) and the data produced by an object that is needed by other project graphical objects (referred to as OUTPUT variables).

Graphically, graphical objects appear as graphical object *icons* in the PARSA Project Explorer. For each INPUT variable an *INPUT port* appears along the top edge of the icon. Similarly, for each OUTPUT variable an *OUTPUT port* appears along the bottom edge of the icon. Hence, the interface properties of a graphical object are represented graphically as ports on the graphical object's icon. An example graphical object icon with 3 INPUT ports and 2 OUTPUT ports is shown in Figure 1.

Figure 1. A PARSA graphical object icon with 3 INPUT ports and 2 OUTPUT ports.

PARSA supports different types of graphical objects that each have unique semantic representations. Regardless of the type of graphical object the interface appears the same. That is, each INPUT variable results in an INPUT port on the icon and each OUTPUT variable results in an OUTPUT port on the icon. To simplify this introductory presentation of PARSA the graphical objects discussed in this section are task graphical objects. The other types of graphical objects and the forms of parallelism they exploit are described later.

A task graphical object, as its name implies, is simply a task to be performed within a project. The task to be performed is programmed by the programmer in a standard programming language. PARSA v2.0 currently supports C and Fortran. This paper presents the PARSA-Fortran. For more information about

Semantically, each task graphical object is an independent, schedulable entity within a project that will be spawned as a thread at run time. As such, multiple graphical objects can be executing concurrently at any given time. Therefore, it is imperative that graphical objects execute without adversely affecting the execution of other graphical objects that may be executing at the same time. Similarly, graphical objects must not be adversely affected by the execution of other graphical objects executing at the same time. The phenomenon where one object (or thread) adversely affects other objects (threads) executing concurrently on a multi-threaded system is known as a *side effect*. The PARSA programming methodology and graphical objects provide an easy-to-use framework for developing side effect free multi-threaded software.

It is intended for graphical objects to be programmed without reference to the target system that will ultimately execute the project, including threading directives. That is, the task performed by a graphical object should be generic code and not include code that is intrinsic to a specific multi-threaded system. There are two important reasons for this:

- First, graphical objects are intended to be computational building blocks that can be re-used within a project and in different projects. If a graphical object is programmed with system-specific code it will limit the re-usability of that object.
- Second, projects developed in PARSA can be easily ported to a wide range of systems if the graphical objects do not contain system-specific code. If even a single graphical object within a project contains system-specific code, the portability of the entire project will be jeopardized.

For these reasons, PARSA programmers are urged to refrain from programming system-specific code into graphical objects. If programmers refrain from embedding system-specific code into their graphical objects then the graphical objects can be re-used between different projects and the projects can be ported to a wide range of systems supported by the ThreadMan Thread Manager [7, 8].

The interaction between graphical objects is specified graphically with *arcs*. Arcs connect source graphical object OUTPUT ports to destination graphical object INPUT ports. Graphically, arcs are simply lines connecting a desired OUTPUT port to a desired INPUT port. Semantically, however, arcs represent data "flowing" or "being passed" from a source graphical object to a destination graphical object. At run time data is passed from the thread of the source graphical object to the thread of the destination graphical object. The PARSA source code generator produces the data structures and code needed to pass data between threads according the graphical configuration of graphical objects and arcs. Figure 2 shows a project in the PARSA Project Explorer with arcs showing the relationship between graphical objects.

Using arcs to specify the relationship between graphical objects eliminates the need for PARSA programmers to generate the code needed to pass data between graphical objects. This is an important feature of the PARSA programming methodology:

- It reduces the amount of code that must be generated by PARSA programmers. This reduces project development time and cost.
- The code to pass data between graphical objects in a multi-threaded system can be sophisticated and complex. Eliminating the need for PARSA programmers to generate data passing code by hand reduces the complexity of developing multi-threaded software.
- The PARSA source code generator is fully automated. Therefore, data passing code is consistently and reliably generated every time project source code is produced. This increases the reliability of software projects developed in PARSA.

4 The PARSA Execution Model

The *PARSA execution model* is this: A graphical object is eligible for execution when all of its INPUT data is available. In other words, a graphical object cannot execute until all the graphical objects that it is dependent upon for data have finished executing and their OUTPUT data is available. If a graphical object does not have any INPUT variables declared (*i.e.*, a graphical object has no INPUT ports), then it is eligible for execution when the project begins executing. Therefore, the graphical representation of a project implicitly defines the order of execution of the graphical objects in the project.

Using this definition it is an easy and informative exercise to determine how the sample project shown in Figure 2 will execute. The graphical object named inputGo has no INPUT ports, and therefore can execute when the project begins executing. Once inputGo has finished executing Go1 and Go2 are eligible for execution. If the system executing the project has multiple processors then Go1 and Go2 can execute concurrently on different processors, and likely will. That is, at run time two threads will be spawned – one for Go1 and one for Go2. Concurrently executing threads reduces the execution time of projects. If the system executing the project has a single processor then Go1 and Go2 will share the processor's cycles until each completes execution. For the purposes of this discussion we assume the system executing this project has multiple processors and Go1 and Go2 will execute concurrently on different processors.

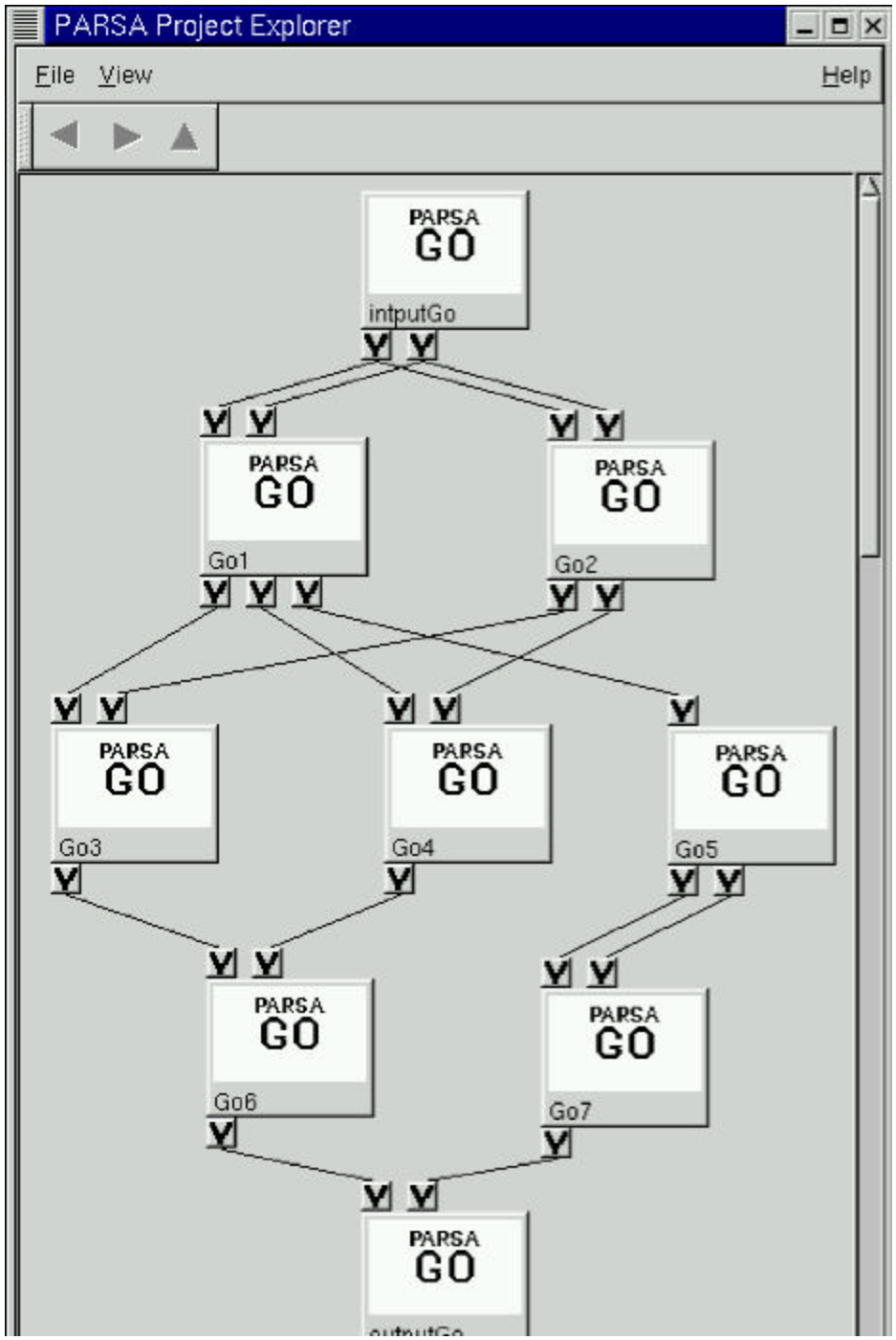
If Go1 executes for a relatively long time compared to Go2, then Go2 will finish executing before Go1. In this case, the OUTPUT data generated by Go2 will be available for Go3 and Go4. However, Go3 and Go4 cannot begin executing because they are dependent on data generated by Go1. Hence, Go3 and Go4 will not be eligible for execution until Go1 finishes executing. Once Go1 finishes executing Go3, Go4 and Go5 are eligible for execution, and they can execute concurrently on separate processors. When Go3 and Go4 finish executing Go6 can then execute, and when Go5 finishes executing then Go7 can execute. Finally, when Go6 and Go7 finish executing outputGo can execute.

On the other hand, if Go2 executes for a relatively long time compared to Go1, then Go1 will finish executing before Go2. In this case, the OUTPUT data generated by Go1 will be available for Go3, Go4 and Go5. Because Go5 is dependent only on data generated by Go1 it is eligible for execution when Go1 finishes executing, and *Go5 will begin executing while Go2 is still executing*. When Go5 finishes executing then Go7 can begin. Only after Go2 has finished executing will Go3 and Go4 be eligible for execution, and Go3 and Go4 must finish executing before Go6 is eligible for execution. Finally, outputGo will be eligible for execution when both Go6 and Go7 have finished executing.

It is interesting to note that this analysis was performed without knowing what tasks each of the graphical objects in this project perform and without knowing exactly what data is passed between the graphical objects. Hence, PARSA's graphical programming methodology aids in the conceptual understanding of projects, which makes software maintenance easier and less cumbersome. Of course, the lower-level details of a project can always be viewed and modified as needed.

It is also interesting to note that PARSA programmers need not concern themselves with the relative execution times of the graphical objects within a project. The PARSA execution model enforces the data dependencies specified with the project's arcs and ensures the graphical objects will execute in the proper order. Therefore, PARSA programmers do not need to manage the coordination of projects or generate the code to do so. As this analysis shows arcs represent not only data being passed between graphical objects but also implicitly control the execution of graphical objects in a project.

5 Graphical Object Types


PARSA Project Explorer
File View Help
PARSA GO
intputGo
PARSA GO
Go1
PARSA GO
Go2
PARSA GO
Go3
PARSA GO
Go4
PARSA GO
Go5
PARSA GO
Go6
PARSA GO
Go7
PARSA GO

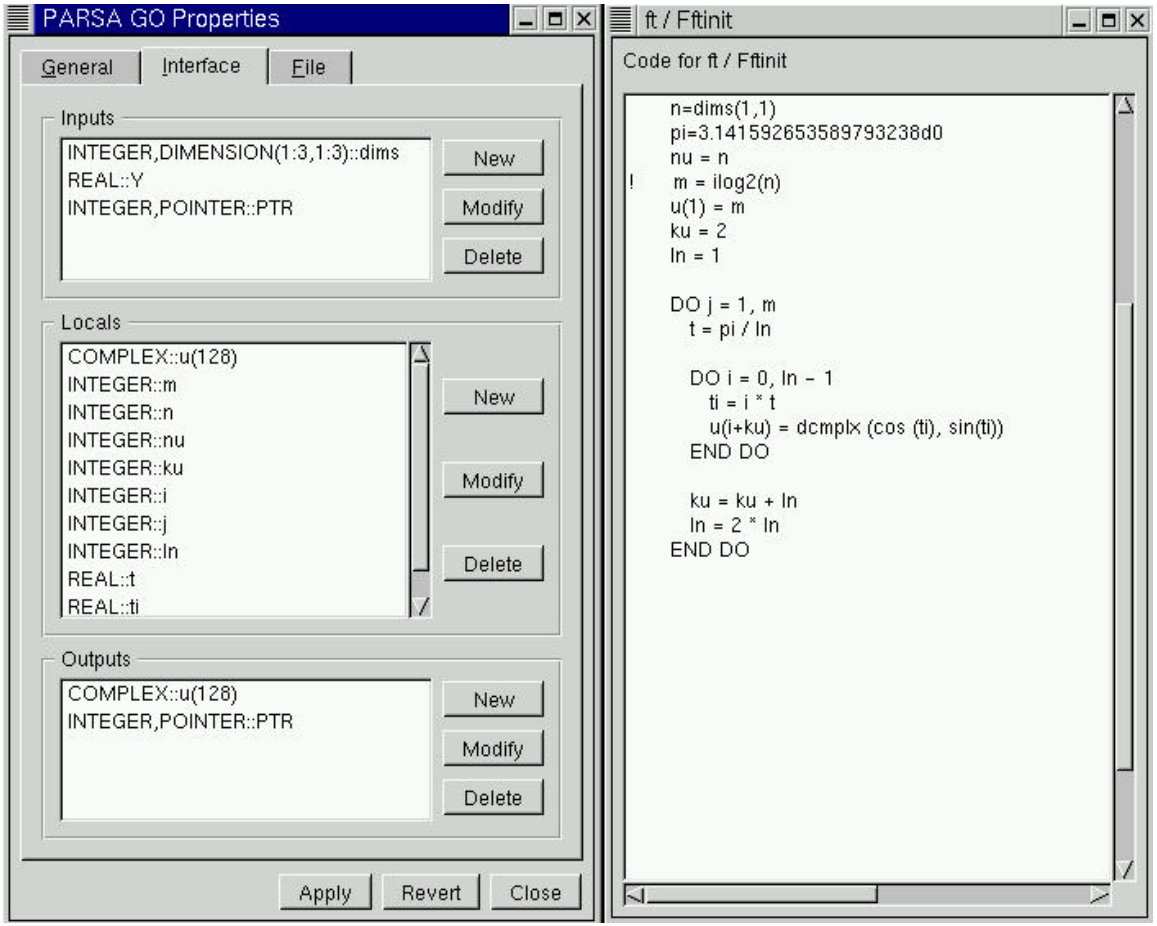


Figure 3. The Task GO Properties Panel and the code associated with a graphical object.

Notice the Interface tab of the Task GO Properties Panel is shown in Figure 3. This is where the interface a task graphical object has with other graphical objects is specified. The interface is simply a collection of variable declarations. Notice the interface has 3 INPUT variables, several LOCAL variables, and 2 OUTPUT variables. The graphical icon for this graphical object would look similar to that shown in Figure 1. Notice the code associated with this graphical object is programmed in Fortran without any threading directives, data passing code, or thread management and coordination code. This makes programming task graphical objects similar to programming subroutines in Fortran.

Because task graphical objects can be programmed to perform any programmer-defined task the granularity of task graphical objects can vary widely from a simple piece of code to a complex and sophisticated routine. Therefore, at run time task graphical objects execute in an irregular manner as the run time analysis of the project shown in Figure 2 illustrated. Hence, task graphical objects allow PARSA programmers to exploit *irregular parallelism* in their software projects.

Task graphical objects can be programmed to perform asynchronous tasks. That is, a task graphical object

5.2 Forall Graphical Objects

Many large-scale commercial and scientific projects spend a large percentage of their time executing loops. To reduce the amount of time spent in loops that execute in a regular manner PARSA supports forall graphical objects. PARSA forall graphical objects allow PARSA programmers to exploit *regular parallelism* in their software projects. Note that regular parallelism is also commonly referred to as data parallelism and loop level parallelism. Throughout this paper regular parallelism is synonymous with data parallelism and loop level parallelism.

In the same modular manner that task graphical objects are programmed, forall graphical objects have programmer-generated code associated with them. The code associated with a forall graphical object is the code that would be in the body of a *DO* loop in Fortran. Forall graphical objects also have properties that define the graphical object name, the interface the graphical object has with other graphical objects in a project, the *parsa_forall* statement, and file properties that control how PARSA stores the information related to the graphical object. The *parsa_forall* statement is similar to the Fortran *DO* statement in that it has three expressions as shown below.

```
parsa_forall e1, e2, e3
```

Notice the keyword *parsa_forall* is used in place of the Fortran *DO* keyword. Expressions e_1, e_2 and e_3 are the forall loop initialization, test and reinitialization expressions, respectively. The *parsa_forall* statement expressions define the number of threads that get spawned at run time. That is, at run time the *parsa_forall* expressions are used to spawn multiple threads, where each spawned thread will execute the body code associated with a forall graphical object.

At run time each spawned thread is passed a unique value of the forall loop control variable (the left hand side of the *parsa_forall* statement expressions). PARSA programmers can reference the loop control variable in the body code to programmatically control which regions of INPUT and OUTPUT arrays are accessed and modified by each thread in the same way loop control variables are used in the body of sequential *DO* loops. Notice that PARSA forall graphical objects do not require programmers to specify how data is distributed to regularly parallel sections of code (such as is supported by High Performance Fortran). Rather, array data is passed to the body threads *by reference*, and the loop control variable is used programmatically to ensure each thread accesses and modifies unique, non-overlapping regions of the arrays. Scalar values are passed to the body threads *by value*, and each thread has a local copy of scalar INPUT and OUTPUT variable.

It should be emphasized that the body code associated with a forall graphical object must perform independent operations. The requirement that allows regular parallelism within a project to be exploited is that each iteration of a loop must be independent of all other iterations of the looping construct. That is, each iteration of a loop must not rely on the results generated by a previous iteration of the loop. Regular parallelism is most commonly found in looping structures that operate on and produce array data, database records and individual files.

The example presented in Section 9 demonstrates the details of programming forall graphical objects.

The PARSA source code generator converts forall graphical objects into source code that will spawn the number of threads defined by the *parsa_forall* statement. That is, the *parsa_forall* expressions determine the number of threads that will be spawned at run time. The threads will safely execute concurrently on a shared memory system improving the run time performance of the looping structure.

hierarchy to view and modify the graphical objects and arcs encapsulated by a composite graphical object. Because composite graphical objects add hierarchy to projects, composite graphical objects provide support for exploiting *hierarchical parallelism*. The PARSA GO Browser provides a useful way to view project hierarchy as shown in Figure 4.

Figure 4. The PARSA GO Browser.

Notice that the composite graphical object foo encapsulates multiple graphical objects (Go1 and Go2). At the highest level of the project hierarchy only the composite graphical object foo is visible. The composite graphical object foo itself encapsulates other graphical objects that can be viewed and modified by double clicking on the composite graphical object as shown in Figure 5.

The graphical objects CompositeInputs and CompositeOutputs are encapsulated by every graphical object and act at the conduit for passing data between the hierarchical levels of projects. Composite graphical objects can encapsulate any of the graphical object types supported by PARSA. That is, composite graphical objects can exploit all forms of parallelism that PARSA exploits, but in a hierarchical manner.

Arcs are used to specify the relationship between graphical objects encapsulated by a composite graphical object in the same manner described above. Arcs in composite graphical objects specify data being passed between graphical objects encapsulated by a composite graphical and implicitly control the order that the graphical objects will execute.

The PARSA source code generator produces the code needed to support hierarchical parallelism. The ThreadMan Thread Manager manages the execution of the graphical objects encapsulated by a composite graphical object. Each graphical object encapsulated by a composite graphical object will execute according to the semantic representation of its graphical object type.

5.4 While Graphical Objects

The PARSA while graphical object is a special type of composite graphical object that allows PARSA programmers to easily exploit *repeat parallelism* in their projects. The while graphical object supports repeat parallelism by encapsulating a collection of graphical objects in the same way composite graphical

While graphical objects can encapsulate any of the graphical object types supported by PARSA. That is, the body of a while graphical object can include task, forall, composite and while graphical objects. Arcs are used to specify the relationship between the graphical objects in the body of a while graphical object.

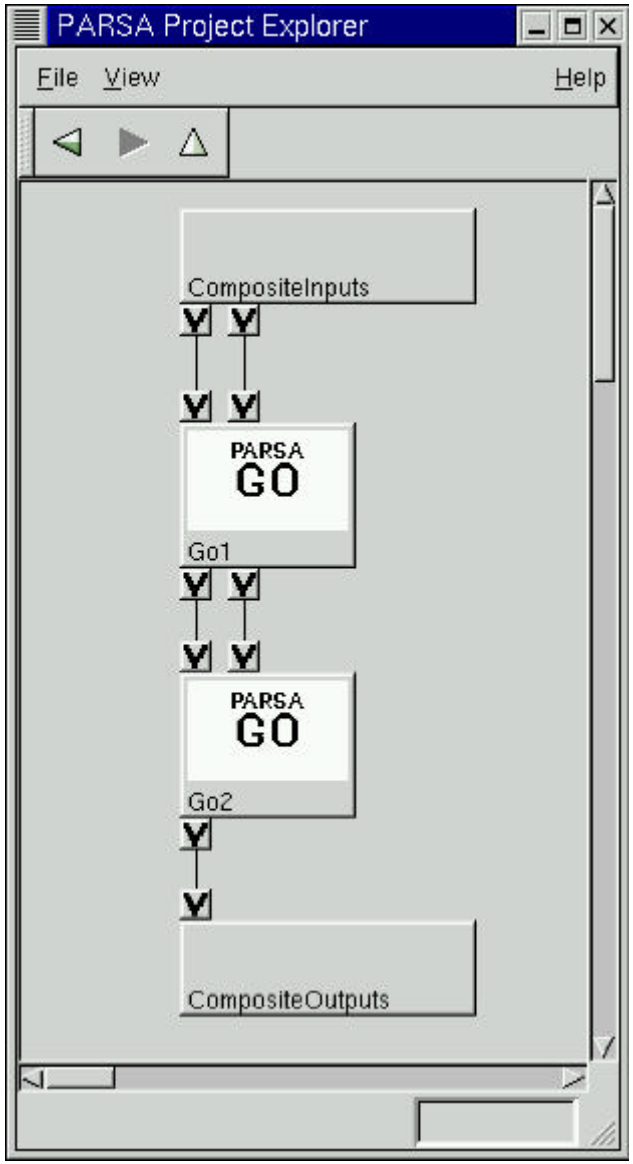


Figure 5. The "inside" view of a composite graphical object.

Arcs represent data being passed between graphical objects and control the order that the graphical objects

Notice that dependencies *can* exist between loop iterations. Therefore, the order of execution of loop iterations must be maintained. The data that is modified between loop iterations that is needed in subsequent iterations are specified as UPDATE variables on the Interface tab of the While GO Properties Panel. Figure 6 shows the While tab of the While GO Properties Panel and the inside view of a while graphical object in the PARSA Project Explorer.

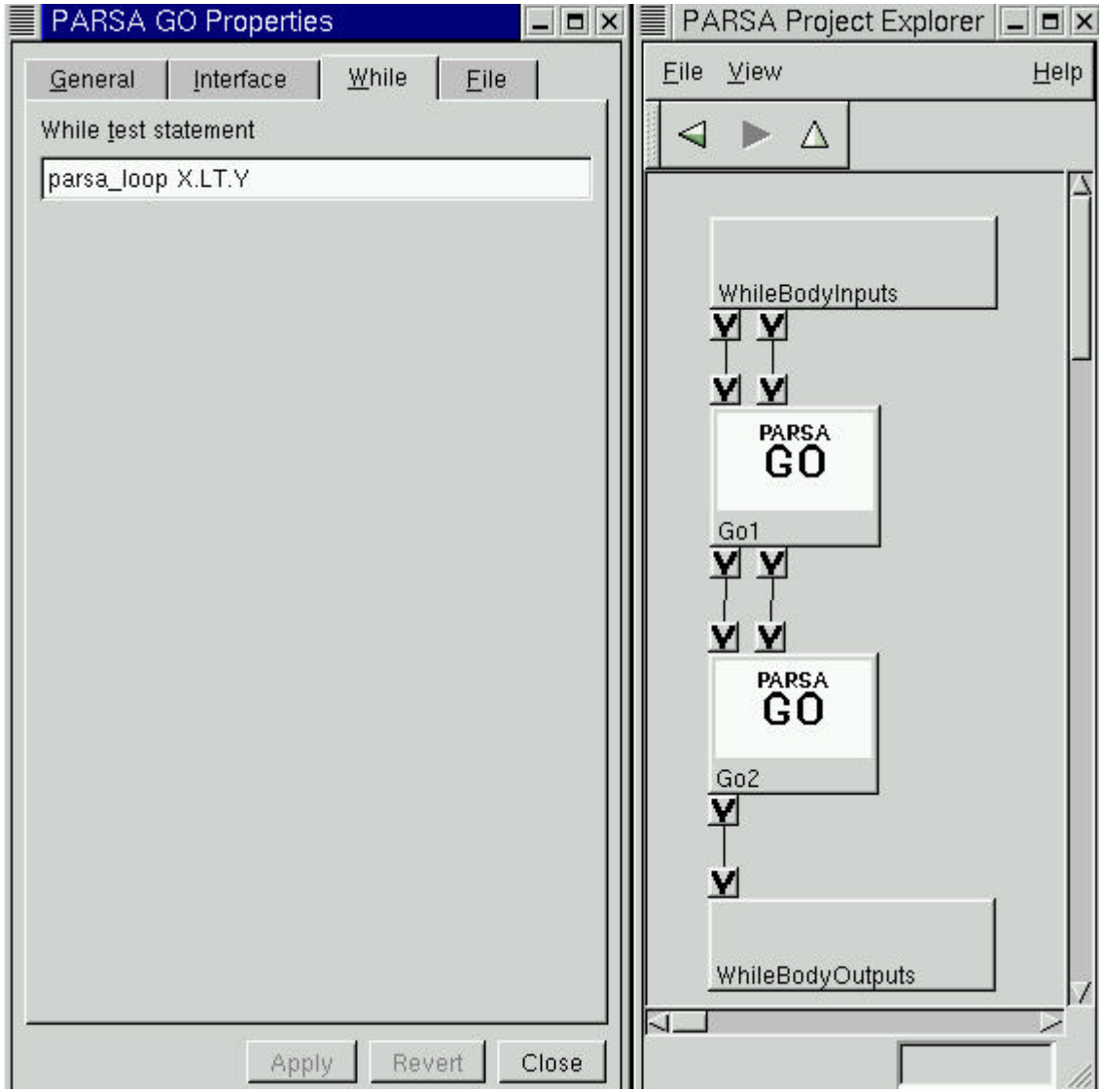


Figure 6. The While GO Properties Panel and the inside view of a while graphical object.

The PARSA source code generator produces source code that supports the semantic representation of while

all forms of parallelism exploitable by PARSA, and PARSA programmers must determine which forms of parallelism can be exploited in their software projects. However, the PARSA programming methodology allows programmers to easily exploit these forms of parallelism to create scalable, reliable and efficient multi-threaded software.

6 Programming a Multi-Threaded Fortran Project

To program a multi-threaded Fortran project in PARSA the File-New option is selected. A project wizard executes that prompts PARSA programmers to select the project type (currently PARSA supports application development and multi-threaded function development) and language (currently PARSA supports C and Fortran). A blank PARSA Project Explorer pops up. To add graphical objects to a project CLICK on the desired type of graphical object, CLICK on the PARSA Project Explorer canvas, and a graphical object icon will be added to the canvas.

To program the code associated with a graphical object (task and forall graphical objects) double CLICK the mouse – a code programming panel will popup. The code associated with the graphical object is programmed in the code programming panel in the specified language (in the case of this paper, Fortran). To add graphical objects to composite and while graphical objects double CLICK the mouse button and the PARSA Project Explorer will expose the inside of the graphical object. Graphical objects are added to composite and while graphical object in the same manner discussed above. Graphical objects encapsulated by composite and while graphical objects are programmed in the same manner discussed above. That is, there are no special rules for programming graphical objects encapsulated by composite and while graphical objects.

To specify the properties of a graphical object CLICK the right mouse button and select the “Properties” option from the popup menu. This pops up the GO Properties Panel for the type of graphical object being edited. For brevity, only the interface properties will be presented here. The interface properties of graphical objects consist of variable declarations for each piece of data that will be passed to or passed from a graphical object. The interface properties are programmed as variable declarations in the same manner variables are declared in a Fortran subroutine. Therefore, specifying the interface properties is as simple as making variable declarations.

Arcs are added with a left mouse CLICK on a source graphical object OUTPUT port, dragging the mouse to a destination graphical object INPUT port, CLICK the left mouse button, and the arc will be drawn.

Thus far the mechanical aspects of generating Fortran projects in PARSA have been covered. Another important aspect of developing projects in PARSA is the logical partitioning of projects in preparation for using PARSA. The following is a list of issues that PARSA programmers should address to partition their projects.

- Identify coarse-grained operations. If an application can defined as a collection of coarse-grained operations that interact with each other then each operation should be specified in PARSA as a composite graphical object. Notice that at this early stage of development PARSA programmers do not have to deal with the lower level issues of exactly what types and how many graphical objects will be encapsulated by the composite graphical objects. Those details can be added later.

- Identify regular parallelism. Regular parallelism should be exploited whenever possible because of its potential to increase project performance. Application loops should be investigated to determine if they contain regular parallelism. If an application contains regular parallelism, or it can be modified such that it contains regular parallelism, then a forall graphical object should be added to the project. Again

graphical objects in PARSA so they will be executed concurrently at run time. A task graphical object should be added to the project for each task identified.

Once the analysis has been performed lower level details can be added to the graphical objects. For example, the name and interface properties might be specified for all graphical objects at the highest level in the project hierarchy. Once the interface properties are defined the relationship between graphical objects can be specified by adding arcs between graphical objects. The project composite and while graphical objects can be analyzed in more detail to determine what types of graphical objects will be contained within each. The code associated with task and forall graphical objects can be added whenever it makes sense to do so. This progressive, iterative approach continues until all project details have been specified.

Notice that the iterative approach suggested above is similar to generating an application flowchart. However, the hierarchal parallelism supported by PARSA requires a third dimension to be added to the flow chart, which can be easily viewed with the PARSA GO Browser. The generation of a project flowchart can be written on paper and translated into PARSA or directly put into PARSA. Graphical objects can always be added to a project, existing graphical objects can be removed, and the properties and code associated with graphical objects can be modified at any time. Therefore, PARSA programmers can use PARSA to rapidly prototype applications using an evolutionary programming approach to incrementally fill in project details over time.

7 The PARSA Source Code Generator and The ThreadMan Thread Manager

Once a project has been specified the graphical representation is converted into multi-threaded source code by the PARSA source code generator. The programmer-generated graphical object code is used as the basis of the code produced by PARSA, and is augmented with all structure declarations, data passing code, threading directives and code needed by the ThreadMan Thread Manager to manage project execution according to the PARSA execution model.

The PARSA source code generator produces two sets of code for Fortran projects. Figure 7 shows a graphical depiction of the source code produced by the PARSA source code generator for Fortran projects and how it interacts with the ThreadMan Thread Manager.

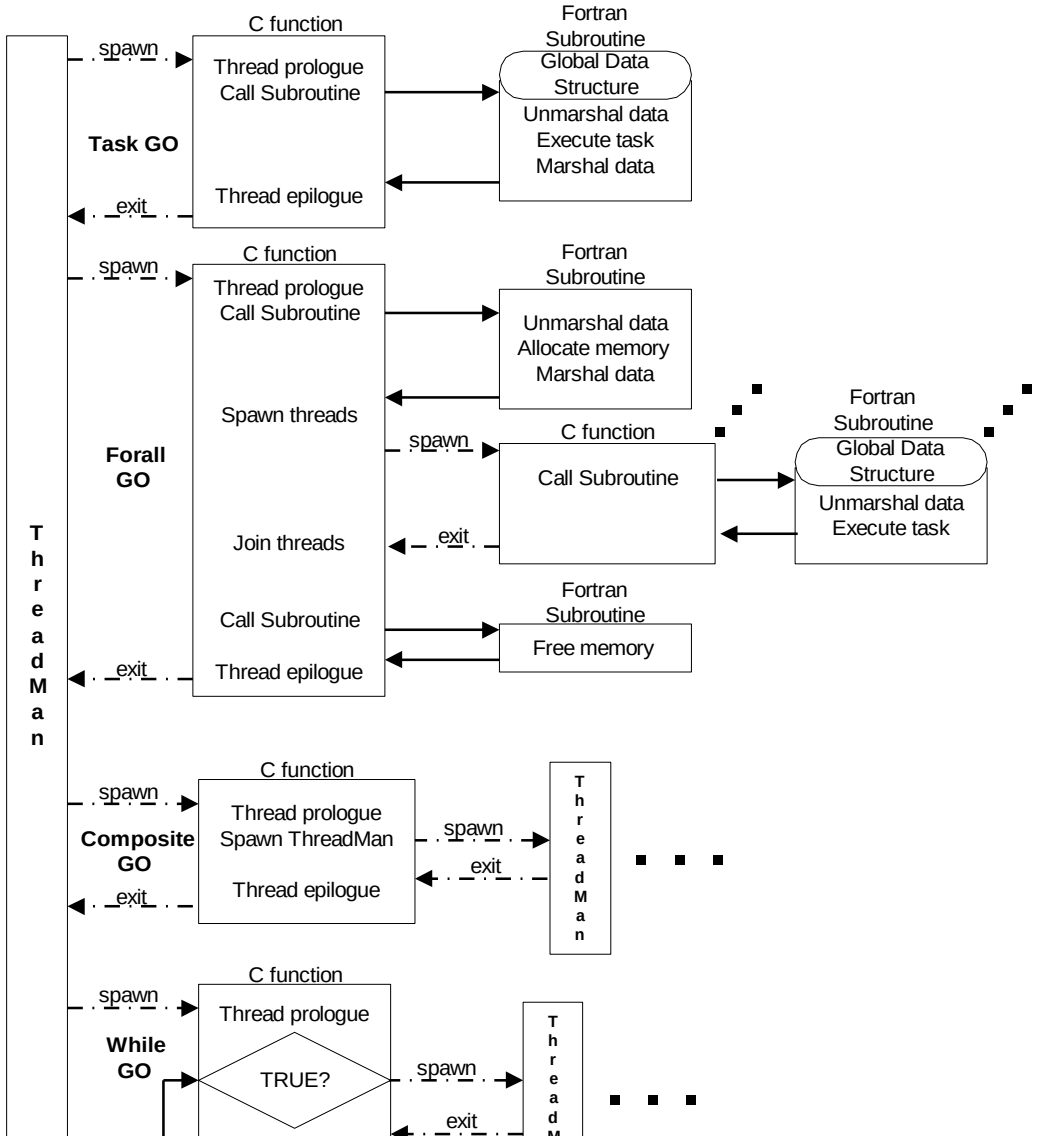
The first thing to notice is that projects developed in PARSA are managed at run time by ThreadMan. ThreadMan monitors the state of a project and determines when graphical objects are eligible for execution. When a graphical object becomes eligible for execution (*i.e.*, all INPUT data is available) ThreadMan spawns the C function (as a thread) associated with the graphical object. Notice that dashed lines indicate where threads are spawned and exit. The type of graphical object determines what happens next.

Task graphical objects have a C function and a Fortran subroutine generated by the PARSA source code generator. The C function is spawned as a thread by ThreadMan, and executes thread prologue operations, calls the Fortran subroutine, and executes thread epilogue operations. The C function is a control mechanism for managing the execution of the Fortran subroutine within the PARSA execution model. The Fortran subroutine for a task graphical object unmarshals INPUT data (*i.e.*, assigns INPUT data to local variable references), executes the programmer-defined task, and marshals (*i.e.* passes) OUTPUT data to successor graphical objects. Notice that each task graphical object has a global data structure allocated by the PARSA source code generator for passing data between graphical objects.

Also notice that all data passing is done by Fortran code eliminating impedance mismatches between C and Fortran. Doing all data passing in Fortran eliminates compiler compatibility issues between different combinations of C and Fortran compilers, which are difficult to support universally across a wide range of

is spawned as a thread by the first C function, and acts as the control mechanism for the forall task subroutine in a manner similar to task graphical objects.

Composite graphical objects have a C function generated by the PARSA source code generator. The C function performs thread prologue operations, spawns ThreadMan to manage the execution of graphical objects encapsulated by the composite graphical object, and performs thread epilogue operations. When all graphical objects encapsulated by a composite have finished executing program control returns to the C function. Notice that each level of project hierarchy results in ThreadMan being invoked at run time.



is called repeatedly so long as the loop control condition is true. Each level of project hierarchy results in ThreadMan being invoked at run time.

As Figure 7 shows the PARSA source code generator produces much of the code needed for Fortran projects to be multi-threaded. By abstracting programmers from the low-level details of multi-threaded programming PARSA frees programmers to concentrate on the "what" of their projects, not the "how." This is especially important to Fortran programmers who are typically domain experts in technical fields, not computer scientists. Hence, Fortran programmers can use PARSA to develop efficient and scalable multi-threaded Fortran projects quickly and easily.

ThreadMan is a dynamic linkable library that manages the execution of projects developed in PARSA as shown in Figure 7. ThreadMan is an integral component of the PARSA programming methodology that i.) eliminates the need for PARSA programmers to control the execution of their software projects, ii.) ensures projects execute according to the PARSA execution model and iii.) makes project source code portable across a wide range of hardware platforms and operating systems supported by ThreadMan. The source code produced by the PARSA source code generator is fully compatible with run time management functions in the ThreadMan library.

As Figure 7 illustrates, ThreadMan is used in numerous situations to control the execution of project graphical objects. That is, ThreadMan is invoked for each level of project hierarchy. Therefore, ThreadMan is a critical component of PARSA's ability to support hierarchical programming and exploiting hierarchical parallelism.

What is not shown in Figure 7 and should be emphasized is that ThreadMan will spawn threads as they become eligible for execution. ThreadMan can, and usually will, have multiple threads under its control at any given time. That is, ThreadMan can concurrently manage multiple threads. This feature produces the run time performance gains for projects developed in PARSA.

Notice too that ThreadMan makes projects *automatically scalable*. That is, the same project will scale to utilize the resources of different systems, and is achieved without the need to recompile project source code. A project can be compiled once and executed on different systems with varying numbers of processors installed. For example, a project can be compiled for one operating system and executed on different systems running that operating system. The number of processors installed on each system will dictate the run time performance of the project. When executed on a single processor system PARSA projects typically run in the same amount of time as an equivalent sequential application. When the project is executed on a multiprocessor system the project will automatically scale to take advantage of the resources of that system. This allows PARSA programmers to develop projects on development systems and deploy those projects on more powerful deployment systems without the need to recompile the project. Automatic scalability is facilitated by ThreadMan.

The code produced by the PARSA source code generator can be easily ported to systems from different vendors by re-compiling the code for each desired system linking the ThreadMan library for each system. This allows projects developed in PARSA to be deployed on systems that employ drastically different threading mechanisms. The most extreme difference between threading mechanisms is between those supported by the Microsoft Windows operating system and those supported on Unix-based systems. The threading mechanisms supported by these operating systems are very different syntactically and semantically. ThreadMan allows projects developed in PARSA to be easily ported between systems as diverse as Windows and Unix. Hence, abstracting the PARSA-generated source code from native threading

8 Embedding Threading Directives Into Projects

There are certain situations when programmers may need or want to embed threading directives into their Fortran projects being developed in PARSA. The Fortran-Pthreads API from Gabb et al can be used to embed threading directives into Fortran projects without devising a scheme to directly access the native threading directives. The Fortran-Pthreads API provides Fortran-like access to the majority of the native pthreads calls. The Fortran-Pthreads API is available on a wide range of systems ensuring project portability.

9 An Example Project

To demonstrate how projects are developed in PARSA a simple example is presented. The example, matrix multiplication, was chosen because of the universal knowledge of the algorithm. Choosing a simple algorithm allows the presentation to focus on how Fortran projects are developed in PARSA without getting lost on the details of a sophisticated algorithm. However, such a simple example does not demonstrate all the features and functionality of PARSA. Those interested in a more detailed presentation should visit www.PrismPTI.com or contact the authors.

Calculating P(X, Z) = A(X, Y) * B(Y, Z), where A, B and C are arrays, requires three nested *DO* loops as shown below.

```
DO I = 1, X
      DO J = 1, Z
            P(I, J) = 0
            DO K=1, Y
                  P(I, J) = P(I, J) + A(I, K) * B(K, J)
            END DO
      END DO
END DO
```

Analyzing this code reveals that the outermost loop contains regular parallelism. That is, the body of the outer loop calculates a row of matrix P, and each row of matrix P is calculated independent of the calculations of all other rows. Hence, a forall graphical object can be used to exploit the regular parallelism of the outer DO loop.

We assume the forall graphical object that will perform the matrix multiplication is part of a larger project. The arrays A, B, and P will be passed to the forall graphical object along with the array dimensions X, Y and Z. That is, arrays A, B, P and dimensions X, Y and Z will be INPUT variables to the forall graphical object. The forall graphical object will populate array P, and will pass array P and dimensions X and Z to another graphical object in the project. Hence, array P and dimensions X and Z will be OUTPUT variables of the forall graphical object. The interface tab of the Forall GO Properties Panel is shown in Figure 8. Notice the forall graphical object needs three loop control variables I, J and K, which are declared as LOCAL variables.

Notice, the code associated with the forall graphical object is identical to the body code of the outermost DO loop shown above. The body code of forall graphical objects is the code contained within the body of the corresponding DO loop being replaced by the forall loop.

The *parsa_forall* statement simply replaces the DO loop as shown below:

```
parsa_forall(I = 1; I.LE.X; I = I+1)
```

Array P, however, is being populated by each thread, and therefore, warrants closer scrutiny to determine if this is can be done safely. Figure 9 shows the access pattern for each thread. When I is 1 its corresponding thread populates row 1 of array P, when I is 2 its corresponding thread populates row 2 array P, and so on. Therefore, it is clear that each spawned thread populates a unique, non-overlapping region of array P. Therefore, a forall graphical object programmed with the *parsa_forall* statement above will execute safely.

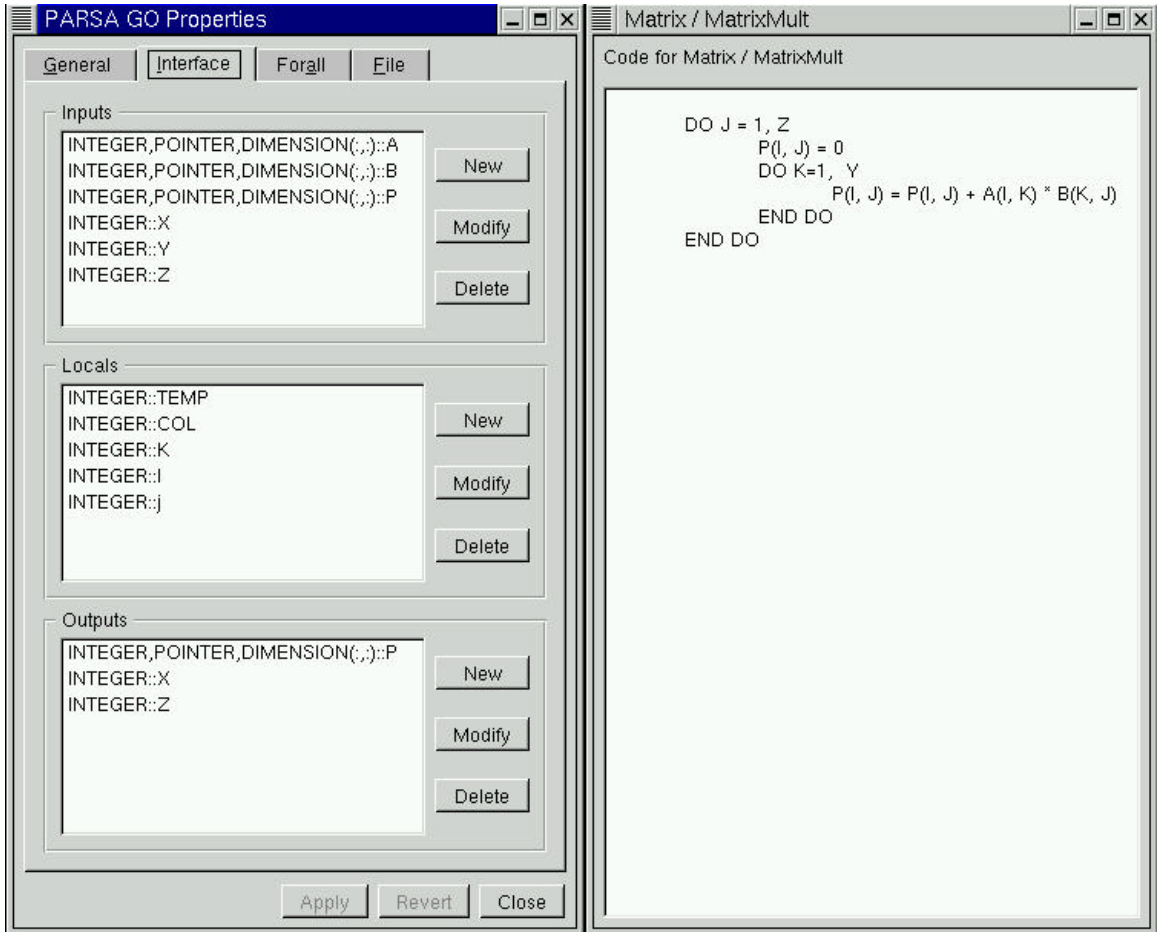
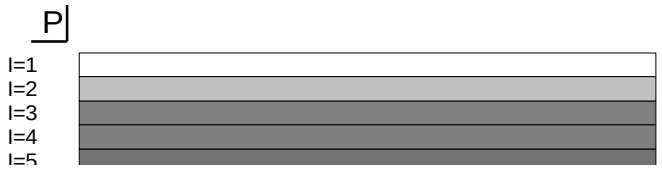


Figure 8. The Interface tab of the Forall GO Properties Panel and code programming panel for a forall



number of system processors. The actual performance will vary depending on the granularity of the forall body code and the loading on the system, but our tests reveal that near linear performance is achievable.

To complete this example project two task graphical objects are added to the project. The first task graphical object, called AllocAndPopulate, allocates memory for arrays A, B and P based on dimensions X, Y and Z and populates arrays A and B with random numbers. Arrays A, B and P and dimensions X, Y and Z are passed as OUTPUT variables from AllocAndPopulate. Figure 10 shows the code associated with AllocAndPopulate alongside the Interface tab of the Task GO Properties Panel.

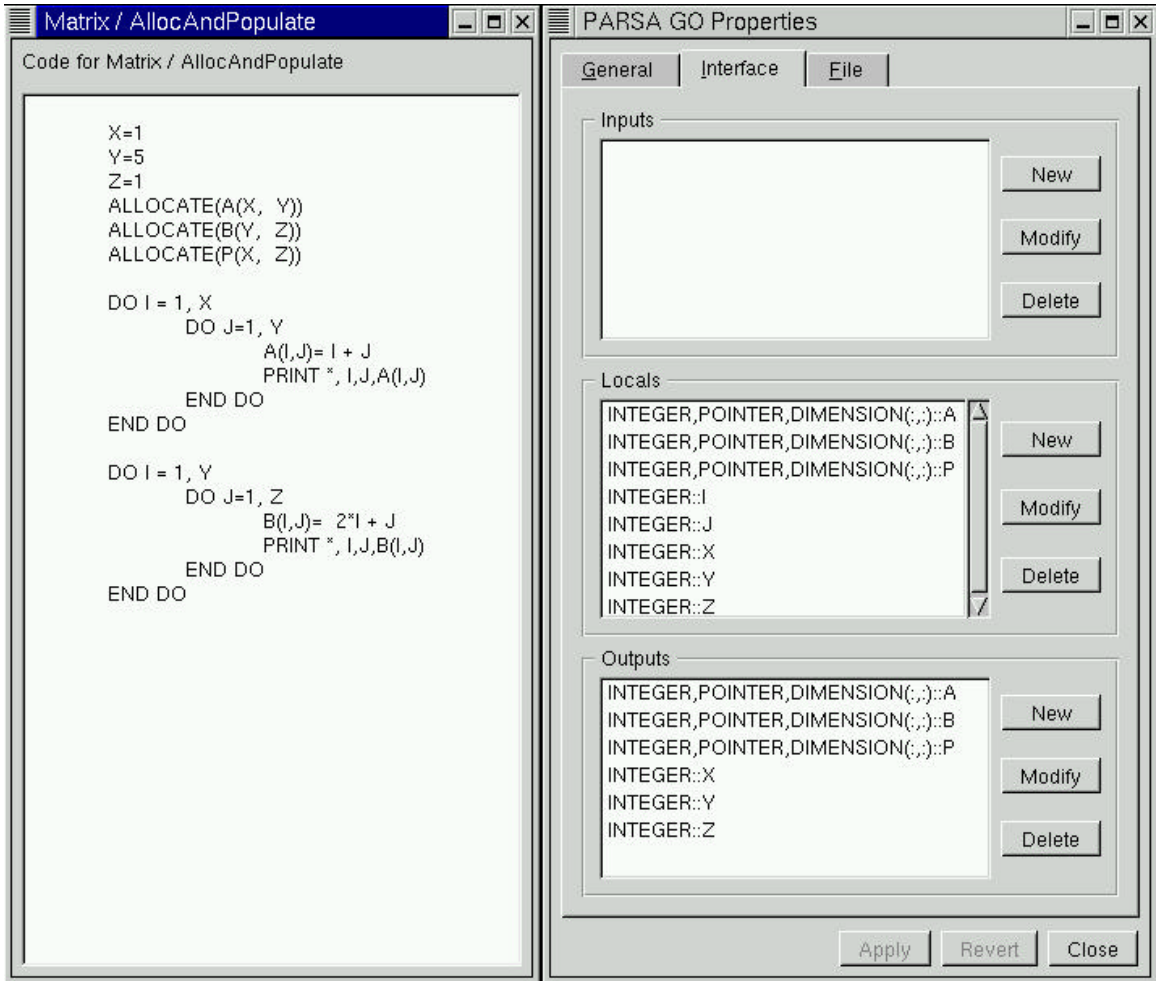


Figure 10. The programmer-generated code for AllocAndPopulate alongside the Interface tab of the Task GO Properties Panel.

The second task graphical object, called DisplayResults, simply prints the values of array P after the forall has populated the elements. Therefore, DisplayResults receives array P and dimensions X and Z. Figure 11 shows the code associated with the DisplayResults and the Interface tab of the Task GO Properties Panel.

The PARSA source code generator produces the multi-threaded source code for this project. The PARSA-generated source code is compiled and ThreadMan linked to manage the execution of the graphical objects at run time.

It is interesting to note that this example was generated simply and easily in PARSA. The code associated with each graphical object is pure Fortran code with the *parsa_forall* statement replacing an equivalent *DO* statement. The resultant project will automatically scale to take advantage of the resources of a target shared memory system, but the programmer is not required to know how this is accomplished. The

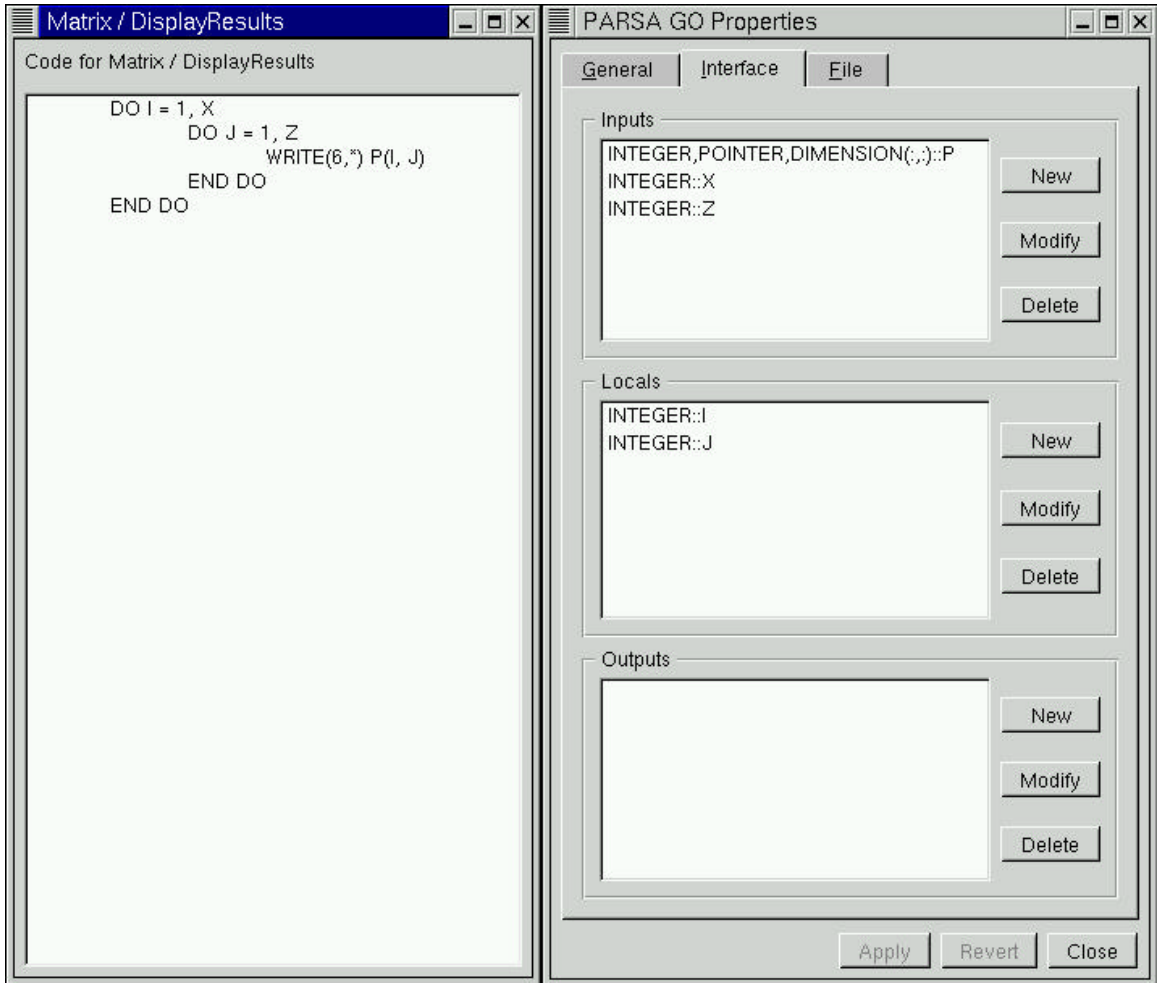
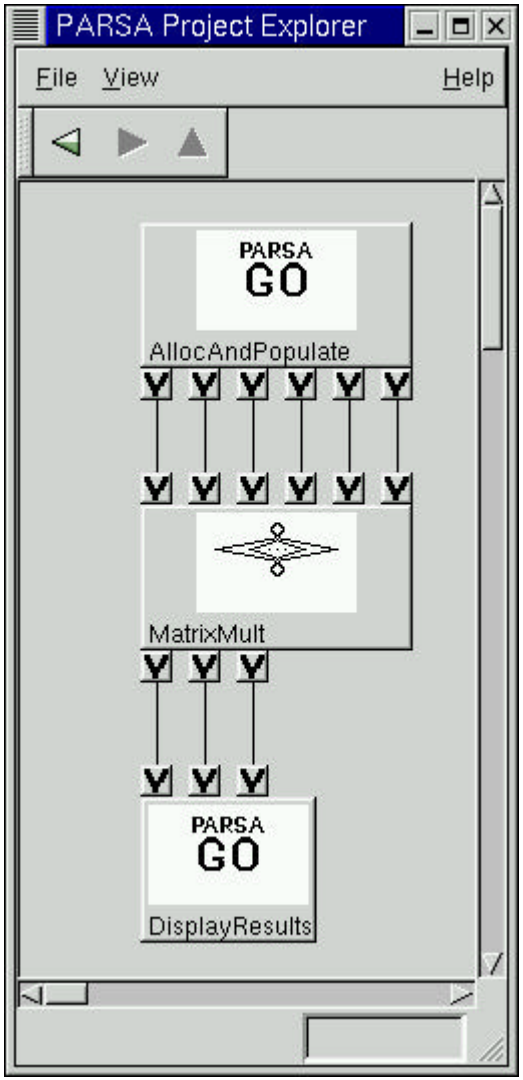


Figure 11. The programmer-generated code for DisplayResults alongside the Interface tab of the Task GO Properties Panel.

programmer will simply have a project that scales to utilize the resources of shared memory systems with little more programming effort than required to program an equivalent sequential application.

stretching development cycles, and requiring programmers to have programming expertise in the use of threading mechanisms. The PARSA programming methodology was then shown to be an efficient method for developing multi-threaded software without imposing additional development requirements on programmers. Therefore, programmers can use PARSA to exploit the scalable run time performance benefits of multi-threaded software without additional development cost, time or effort.

The PARSA Software Development Environment was shown to be an effective method for exploiting many different types of parallelism contained within projects. Specifically, it was shown that PARSA supports irregular, asynchronous, regular, hierarchical and repeat parallelism by supporting different types



12 References

[1] H.A. Gabb, R.P. Bording, S.W. Bova, C.P. Breshears, "A Fortran 90 Application Programming Interface to the POSIX Threads Library," *40th Cray User Group Conference Proceedings*, 1998.

[2] C.P. Breshears, H.A. Gabb, S.W. Bova, "Towards a Fortran 90 Interface to Pthreads," *DoD User Group Proceedings*, 1998.

[3] C.P. Breshears, M.R. Fahey, H.A. Gabb, "Application of Fortran Pthreads to Linear Algebra and Scientific Computing," *Cray User Group*, 1999.

[4] *The PARSA Software Development Environment v2.0 Programming and Reference Manual For the Fortran Programming Language*, Prism Parallel Technologies, Inc., 2001.

[5] *The PARSA Software Development Environment v2.0 – Executive Summary*, Prism Parallel Technologies, Inc., 2001.

[6] *The PARSA Software Development Environment v2.0 – A Technical Primer*, Prism Parallel Technologies, Inc., 2001.

[7] *The ThreadMan Thread Manager v2.0 – Executive Summary*, Prism Parallel Technologies, Inc., 2001.

[8] *The ThreadMan Thread Manager v2.0 – A Technical Primer*, Prism Parallel Technologies, Inc., 2001.